



UNIVERSITÀ DI PISA

*Computer Engineering*

*Advanced Network and Wireless Systems*

**"Automated Network Configuration and Testing with  
Containerlab and GUI Integration"**

Academic Year: 2024/2025

Authors:

*Francesco Taverna*

*Gabriele Pianigiani*

*Saverio Mosti*

## Table of Contents

1. Introduction.....	4
1.1 Purpose of the Project .....	4
2. Project Specifications.....	5
2.1 Objectives.....	5
2.2 Tasks Overview .....	5
2.3 Expected Outcomes.....	5
3. System Architecture .....	6
3.1 Overview of the Architecture .....	6
3.2 Tools and Technologies Used.....	6
3.3 Project Folder Structure .....	7
4. Network Configuration Automation .....	9
4.1 Configuration File Structure .....	9
4.2 Automating DHCP and BGP Rules.....	13
4.2.1 DHCP Automation .....	13
4.2.2 BGP Automation .....	13
4.3 YAML File Format for Topology Definition and Device Configuration .....	15
4.3.1 Topology Definition .....	16
4.3.2 Device Configuration .....	17
5. Network Deployment Using Containerlab.....	21
5.1 Network Topology Overview .....	21
5.2 Deployment Process.....	21
5.2.1 Main steps .....	21
5.2.2 Detailed description .....	22
5.3 Destroy Process.....	22
5.4 Internet Connectivity via NAT and Default Route Distribution .....	22
6. Automated Network Testing .....	24
6.1 Testing Tools: iperf and tcpdump.....	24
6.2 Testing Scenarios and Results .....	24
7. Graphical User Interface (GUI) .....	27
7.1 Features and Functionalities.....	27
7.2 Workflow Explanation .....	27

7.3 User Guide .....	28
7.3.1 Accessing the GUI .....	28
7.3.2 Performing Deployment.....	28
7.3.3 Performing Routers and Switches configuration .....	29
7.3.4 Performing BGP configuration .....	29
7.3.5 View created topology.....	29
7.3.4 Performing Iperf Test .....	30
7.3.4 Performing Network Destruction.....	32
8. Conclusion .....	33

# 1. Introduction

## 1.1 Purpose of the Project

The purpose of this project is to design and implement a comprehensive solution for automating network configuration, deployment, and testing. Modern network management demands efficient tools and practices to reduce manual effort and ensure scalability. This project focuses on leveraging automation to streamline the processes of generating network configurations, deploying network topologies using containerized environments, and validating network performance through automated testing tools.

## 2. Project Specifications

### 2.1 Objectives

The primary objective of this project is to design and implement an automated system for network configuration, deployment, and testing. The system leverages modern technologies such as container-based virtualization, automation tools, and a web-based graphical user interface (web-GUI) to simplify and enhance the process of managing and testing network infrastructures.

### 2.2 Tasks Overview

The project is divided into three main tasks, as outlined below:

#### 1. Network Configuration Automation

Develop a tool capable of generating configuration files for network devices (routers and switches) based on input YAML files. The configurations will include essential parameters such as IP address for all interfaces, DHCP server setup, BGP neighbors and rules, and AS number. This task aims to reduce manual effort and ensure consistency across the network.

#### 2. Network Deployment Using Containerlab

Implement the deployment of the network topology using Containerlab, a container-based network emulator. The system should read a YAML topology file, deploy the defined nodes (routers, switches, and hosts), and establish the specified links between them. The deployment process should be automated and integrated with the configuration tool.

#### 3. Network Testing

Provide functionalities for testing the deployed network. This includes:

- Running *iperf* for traffic generation and performance measurement between selected hosts.
- Using *tcpdump* to monitor and capture traffic on specific interfaces.
- Ensuring that network devices and configurations (e.g., DHCP-assigned IPs, BGP routes) work as intended.

### 2.3 Expected Outcomes

By completing this project, the following outcomes are expected:

- **Automated Configuration Tool:** An application capable of generating device configuration files based on YAML input.
- **Seamless Network Deployment:** A fully functional network topology deployed on a virtualized environment using Containerlab.
- **GUI for User Interaction:** A user-friendly interface for uploading topology and configuration files, deploying networks, and initiating tests.

## 3. System Architecture

### 3.1 Overview of the Architecture

The architecture of this project is designed to simplify the automation, deployment, and testing of network topologies using modern tools and methodologies.

The system consists of three primary layers:

1. **Input Layer:** YAML files are used to define the network topology, device configurations, and interconnections.
2. **Automation Layer:** Python scripts parse the YAML files and generate device-specific configuration files. In addition, other scripts allow network traffic testing over the created topology.
3. **Deployment Layer:** The virtual network topology is deployed using Containerlab, which manages the creation of network nodes (routers, switches and hosts) and their connections.

### 3.2 Tools and Technologies Used

This project leverages a variety of tools and technologies to achieve its objectives. The primary tools and their roles are listed below:

#### YAML

- Type used to define the network topology and device parameters.
- Provides a human-readable and structured way to represent configurations.

#### Python

- Serves as the primary language for automation.
- Scripts handle:
  - the parsing of YAML files
  - generation of configuration files
  - sending commands to devices and receiving their outputs.

#### Containerlab

- A lightweight tool for deploying container-based network topologies.
- Allows quick instantiation of virtual routers, switches, and links based on the YAML definitions.

#### Iperf

- Command used for performance testing to measure network bandwidth and latency.

## Tcpdump

- A packet analysis tool used for capturing and analyzing network traffic to check connectivity.

## Html, CSS, Javascript

- Web coding languages to offer a GUI for exploiting the provided functionalities

These technologies work together to ensure the efficient execution of the project from automation to testing.

## 3.3 Project Folder Structure

The project is organized into a structured directory to facilitate ease of navigation and maintainability. Below is the folder structure:

*ANAWS\_Project/*

```
|
|
|— automatic_configuration/      # Functions to send command to devices
and interact with them
|   |— configure_hosts_dhcp.py
|   |— iperf_and_tcpdump_command_sender.py
|   └— send_configuration.py
|
|— template/                    # Contains jinja template, txt configuration
files and input yaml
|   |— business_rules/
|   |   |— R1_business_rules.txt
|   |   ...
|   |   └— Rn_business_rules.txt
|   |— configurations/
|   |   |— internet_router_configuration.txt
|   |   |— R1_configuration.txt
|   |   └— SW1_configuration.txt
|   |— network_configuration_and_topology/
|   |   |— business_rules_configuration.txt
|   |   |— network_configuration_task2.yaml
|   |   └— topology_task2.yaml
|   └— create_configuration_file.py
```

```

|   └─ template.j2
|   └─ verification_command.txt
|
|
└─ web_interface/                # Contains all elements to support GUI
    └─ images/                  #Contains useful images for GUI
    └─ script/                  #Contains javascript file to call python scripts
    └─ style/                    #Contains CSS design style
    └─ index.html
    └─ network_topology.json
    └─ server_flask.py
    └─ topology.html
|
└─ README.md                    # Description of the project
└─ requirements.txt             # List of required Python packages and dependencies
└─ yaml_to_json_converter.py     # script to convert yaml to json

```

Each folder serves a specific purpose, ensuring the project is modular and easily extendable.



## 4. Network Configuration Automation

### 4.1 Configuration File Structure

The configuration files generated by the automation scripts are designed to be modular and compatible with common networking device operating systems. Each configuration file includes:

1. **Hostname:** name of the device.
2. **DHCP Settings:** Defines IP address pools and subnet for dynamic IP assignment.
3. **Interface Configuration:** Specifies name, IP addresses (IPv4 and/or IPv6), switchport or not, and subnet mask.
4. **BGP:** Configures AS number, *router\_id* of the router, besides IP address and AS number of neighbors and network between them.
5. **IP routing:** Enable or disable it.
6. **OSPF:** Specifies the id of the OSPF process, the OSPF router id, the network prefixes to be announced and the max number of LSAs (Link-State Advertisements) that the router can handle.

An example configuration snippet for a router generated by the application is shown below:

```
!-----
! GENERAL CONFIGURATION
! -----

hostname R1

! DHCP SERVER CONFIGURATION
dhcp server
    subnet 192.168.1.0/24
    range 192.168.1.2 192.168.1.254

! STATIC CONFIGURATION
no aaa root
management api http-commands
    no shutdown
transceiver qsfp default-mode 4x10G
service routing protocols model multi-agent
spanning-tree mode mstp
system ll
    unsupported speed action error
    unsupported error-correction action error
management api gnmi
    transport grpc default
management api netconf
    transport ssh default
```

```

! -----
! INTERFACES CONFIGURATION
! -----
interface Ethernet1
    no switchport
    ip address 172.20.10.0/31
interface Ethernet2
    no switchport
    ip address 172.20.40.1/31
interface Ethernet3
    no switchport
    ip address 192.168.1.1/24
    dhcp server ipv4
interface Loopback0
    ip address 10.0.0.1/32

! -----
! IP ROUTING CONFIGURATION
! -----
ip routing

! OSPF CONFIGURATION

! BGP CONFIGURATION
router bgp 55001
    router-id 10.0.0.1
    ! default safe policy
    bgp missing-policy direction in action deny-in-out
    bgp missing-policy direction out action deny-in-out
    neighbor 172.20.10.1 remote-as 55002
    neighbor 172.20.40.0 remote-as 55020
    network 192.168.1.0/24

! END CONFIGURATION
!-----

```

This file is computed starting from the following jinja template:

```

!-----
! GENERAL CONFIGURATION
! -----

hostname {{ device.hostname }}

! DHCP SERVER CONFIGURATION
{%- if device.dhcp_server %}
dhcp server
    subnet {{ device.dhcp_server.subnet }}

```

```

        range {{ device.dhcp_server.range }}
{%%- endif %}

! STATIC CONFIGURATION
no aaa root
management api http-commands
    no shutdown
transceiver qsfp default-mode 4x10G
service routing protocols model multi-agent
spanning-tree mode mstp
system ll
    unsupported speed action error
    unsupported error-correction action error
management api gnmi
    transport grpc default
management api netconf
    transport ssh default

! -----
! INTERFACES CONFIGURATION
! -----
{%%- for intf in device.interfaces %}
interface {{ intf.name }}
    {%%- if intf.switch is defined and intf.switch == 0 %}
    no switchport
    {%%- endif %}
    {%%- if intf.ipv4_address %}
    ip address {{ intf.ipv4_address }}{{ intf.ipv4_mask }}
    {%%- endif %}
    {%%- if intf.ipv6_address %}
    ipv6 address {{ intf.ipv6_address }}{{ intf.ipv6_mask }}
    {%%- endif %}
    {%%- if intf.dhcp %}
    dhcp server ipv4
    {%%- endif %}
{%%- endfor %}

! -----
! IP ROUTING CONFIGURATION
! -----
{%%- if device.ip_routing | default('false') %}
ip routing
{%%- endif %}

! OSPF CONFIGURATION
{%%- if device.ospf %}
router ospf
    process-id {{ device.ospf[0].process_id }}
    router-id {{ device.ospf[1].router_id }}

```

```

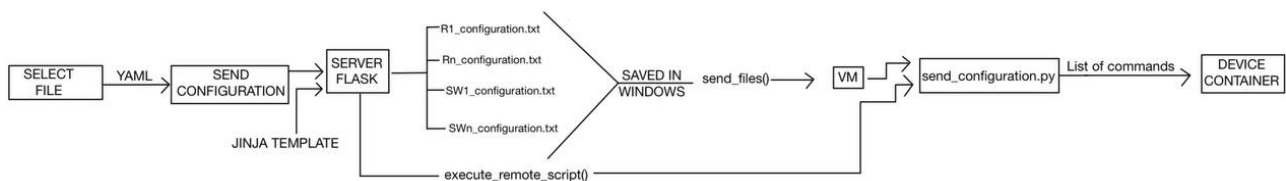
    {%- for net in device.ospf[2].networks %}
    network {{ net.network }}
    {%- endfor %}
    max-lsa {{ device.ospf[3].max_lsa }}
{%- endif %}

! BGP CONFIGURATION
{%- if device.bgp %}
router bgp {{ device.bgp.as_number }}
    router-id {{ device.bgp.router_id }}
    ! default safe policy
    bgp missing-policy direction in action deny-in-out
    bgp missing-policy direction out action deny-in-out
    {%- for neighbor in device.bgp.neighbors %}
    neighbor {{ neighbor.ip }} remote-as {{ neighbor.remote_as }}
    {%- endfor %}
    {%- for net in device.bgp.networks %}
    network {{ net.prefix }}
    {%- endfor %}
{%- endif %}
! END CONFIGURATION
!-----

```

This template is used by the *flask server*, after it received the configuration yaml. A function of the server combines it and the yaml (described later) to create the configuration file shown previously. A different *txt* configuration file is created for each hostname in the yaml, with the following name: *hostname\_configuration.txt*. Then, these files are sent to the VM and a VM remote script is executed to apply all the commands of each file in each device of the containerlab virtual network. To contact containers, a specific function has been used (*get\_containers\_ips*): it gets management IPs from containerlab through *docker inspect* command.

A flow diagram of the configuration process is the following:



It has been necessary to trigger the execution of a remote script on the VM because, due to the VPN layer, windows couldn't work with eAPI that is used to run configuration commands on Arista Cli.

It's important to note that eAPI must be enabled manually before sending the configuration from the web GUI, otherwise it's not possible to communicate with containers (where Arista routers and switches are hosted).

## 4.2 Automating DHCP and BGP Rules

Automation scripts are implemented to execute a DHCP request command on hosts in the network and to apply BGP rules dynamically based on the topology and device settings defined in the YAML files.

### 4.2.1 DHCP Automation

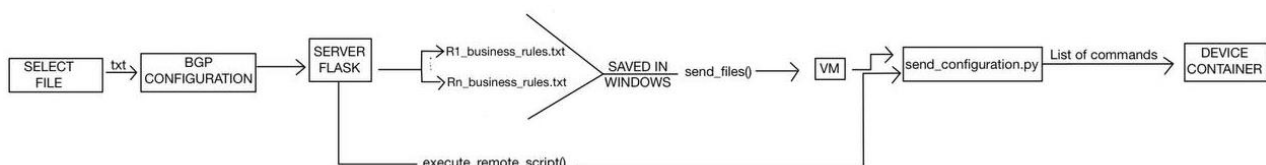
When the configuration file is sent (see previous section) it is also triggered the execution of the remote dhcp script (on the VM). The `configure_hosts_dhcp.py` script allows users to execute this command: `udhcpd -i eth1` on any host specified by name input. To contact host containers, a specific function has been used (`get_host_containers_ips`): it gets eth1 IPs from containerlab through `docker exec ip` command.

Additionally, it runs a series of network commands to properly configure the host, such as installing `iperf` and `tcpdump` commands, inserting the new default route towards the gateway (192.168.x.1).

### 4.2.2 BGP Automation

**Neighbor Relationship Setup:** business relationships must be written in a *txt* configuration file that can be uploaded in the GUI and sent to the *flask server*. That text file must be divided into sections, one for each router, and, at the beginning of each one, there should be the name of the router in this form: `//Rx`. The configuration file is divided into *n* files (*n* is the number of routers), each of them is called: *Rx\_business\_rules.txt* and saved locally. Then, it is followed the same procedure of routers configuration to apply each rule to containers.

A flow diagram of the BGP configuration process is the following:



Task 2 specifies the following requirements to establish business rules between ASes:

- *configure each router to announce all its local networks through BGP;*
- *configure AS55001 as transit for AS55002;*
- *configure AS55020 as transit for AS55001;*
- *configure a peering relation between AS55010 and AS55002;*

Example of a BGP business rules configuration file, developed for the Task 2 completion:

```
//R1
ip prefix-list LAN_R1 seq 10 permit 192.168.1.0/24
ip prefix-list LAN_R2 seq 10 permit 192.168.2.0/24

route-map PERMIT_ALL permit 10

route-map YES_R1_LAN_R1_OUT permit 10
 match ip address prefix-list LAN_R1
route-map YES_R1_LAN_R1_OUT permit 20
```

```

match ip address prefix-list LAN_R2

router bgp 55001
neighbor 172.20.10.1 route-map PERMIT_ALL out // Export all routes to R2.
neighbor 172.20.10.1 route-map PERMIT_ALL in // Import all routes from R2.
neighbor 172.20.40.0 route-map YES_R1_LAN_R1_OUT out // Export only R1 and R2
LANs to R4.
neighbor 172.20.40.0 route-map PERMIT_ALL in // Import all routes from R4.

//R2
ip prefix-list LAN_R2 seq 10 permit 192.168.2.0/24

route-map PERMIT_ALL permit 10

route-map YES_R2_OUT permit 10
match ip address prefix-list LAN_R2

router bgp 55002
neighbor 172.20.10.0 route-map PERMIT_ALL in // Import all routes from R1.
neighbor 172.20.10.0 route-map YES_R2_OUT out // Export only R2 LAN to R1.
neighbor 172.20.50.1 route-map PERMIT_ALL in // Import all routes from R3.
neighbor 172.20.50.1 route-map YES_R2_OUT out // Export only R2 LAN to R3.

//R3
route-map PERMIT_ALL permit 10

router bgp 55010
neighbor 172.20.50.0 route-map PERMIT_ALL in // Import all routes from R2.
neighbor 172.20.50.0 route-map PERMIT_ALL out // Import all routes from R2.

//R4
ip prefix-list LAN_R1 seq 10 permit 192.168.1.0/24
ip prefix-list LAN_R2 seq 10 permit 192.168.2.0/24

route-map PERMIT_ALL permit 10

route-map R4_LAN_INTERNET_OUT permit 10
match ip address prefix-list LAN_R1
route-map R4_LAN_INTERNET_OUT permit 20
match ip address prefix-list LAN_R2

ip prefix-list DEFAULT_ONLY seq 5 permit 0.0.0.0/0
route-map DEFAULT_ONLY permit 10
match ip address prefix-list DEFAULT_ONLY

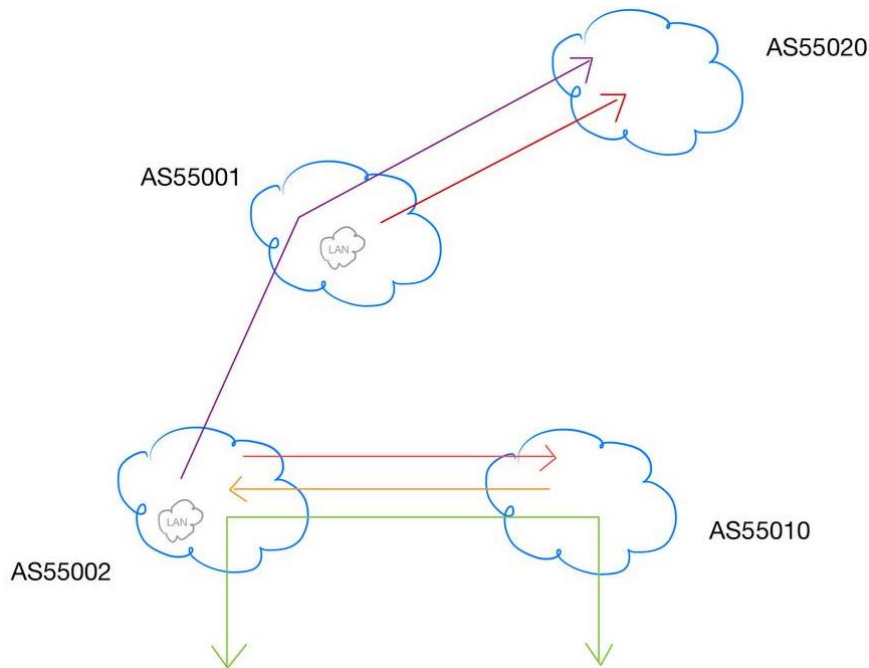
router bgp 55020
neighbor 172.20.40.1 route-map DEFAULT_ONLY out // Export only the default
route to R1.
neighbor 172.20.40.1 route-map PERMIT_ALL in // Import all routes from R1.
neighbor 172.20.100.0 route-map PERMIT_ALL in // Import all routes from
internet_router.
neighbor 172.20.100.0 route-map R4_LAN_INTERNET_OUT out // Export customer LANs
to internet_router.

```

```
//internet_router
route-map PERMIT_ALL permit 10

router bgp 54000
  neighbor 172.20.100.1 route-map PERMIT_ALL out // Export all known routes to R4.
  neighbor 172.20.100.1 route-map PERMIT_ALL in  // Import all routes from R4.
```

The following picture represents which routes are forwarded between ASes to comply with business relationships stated in the project specification:



- **AS55002 → AS55001:**  
Since AS55001 is configured as a transit for AS55002, AS55002 exports all its local networks and its customer routes to AS55001 through BGP.
- **AS55001 → AS55020:**  
AS55020 is configured as a transit for AS55001. Therefore, AS55001 exports all its local networks and customer routes (including those received from AS55002) to AS55020.
- **AS55010 ↔ AS55002:**  
A peering relationship exists between AS55010 and AS55002. This means AS55010 and AS55002 exchange routes for their directly connected local networks and their customers but do not transit traffic from other ASes.

## 4.3 YAML File Format for Topology Definition and Device Configuration

The YAML file serves as the foundation for defining network topologies and device-specific configurations. Its structure is simple yet powerful, enabling scalability for large networks.

### 4.3.1 Topology Definition

The topology section specifies the devices, their roles, and interconnections. Example taken from Task 2 requested topology:

```
name: network_topology

topology:
  nodes:
    # Main routers for each AS
    r1: #AS55001
      kind: arista_ceos
      image: ceos:4.33.0F
    r2: #AS55002
      kind: arista_ceos
      image: ceos:4.33.0F
    r3: #AS55010
      kind: arista_ceos
      image: ceos:4.33.0F
    r4: #AS54020
      kind: arista_ceos
      image: ceos:4.33.0F

    sw1: #AS55001
      kind: arista_ceos
      image: ceos:4.33.0F
    sw2: #AS55002
      kind: arista_ceos
      image: ceos:4.33.0F

    # Devices in LAN of AS55001 (linked to SW1)
    r1_d1:
      kind: linux
      image: alpine:latest
    r1_d2:
      kind: linux
      image: alpine:latest

    # Devices in LAN of AS55002 (linked to SW2)
    r2_d1:
      kind: linux
      image: alpine:latest
    r2_d2:
      kind: linux
      image: alpine:latest

  internet_router:
    kind: arista_ceos
```



```

    image: ceos:4.33.0F

internet_host:
  kind: linux
  image: alpine:latest

links:
  # Links between 4 routers (main backbone)
  - endpoints: ["r1:eth1", "r2:eth1"]
  - endpoints: ["r2:eth2", "r3:eth1"]
  - endpoints: ["r3:eth2", "r4:eth1"]
  - endpoints: ["r4:eth2", "r1:eth2"]

  # LAN of AS55001 (R1)
  - endpoints: ["r1:eth3", "sw1:eth1"]
  - endpoints: ["sw1:eth2", "r1_d1:eth1"]
  - endpoints: ["sw1:eth3", "r1_d2:eth1"]

  # LAN of AS55002 (R2)
  - endpoints: [ "r2:eth3", "sw2:eth1" ]
  - endpoints: [ "sw2:eth2", "r2_d1:eth1" ]
  - endpoints: [ "sw2:eth3", "r2_d2:eth1" ]

#internet
  - endpoints: ["r4:eth3", "internet_router:eth1"]
  - endpoints: ["internet_router:eth2", "internet_host:eth1"]

```

### 4.3.2 Device Configuration

The device configuration section includes details such as IP addresses, BGP settings, and other parameters. Example:

```

devices:
  - hostname: R1
    dhcp_server:
      subnet: 192.168.1.0/24
      range: 192.168.1.2 192.168.1.254
    interfaces:
      - name: Ethernet1
        switch: 0 # no switchport
        ipv4_address: 172.20.10.0
        ipv4_mask: /31
      - name: Ethernet2
        switch: 0 # no switchport
        ipv4_address: 172.20.40.1
        ipv4_mask: /31
      - name: Ethernet3
        switch: 0 # no switchport
        ipv4_address: 192.168.1.1
        ipv4_mask: /24

```

```

    dhcp: true
  - name: Loopback0
    ipv4_address: 10.0.0.1
    ipv4_mask: /32
ip_routing: true
bgp:
  as_number: 55001
  router_id: 10.0.0.1
  neighbors:
    - ip: 172.20.10.1    # R2 - Ethernet1
      remote_as: 55002
    - ip: 172.20.40.0    # R4 - Ethernet2
      remote_as: 55020
  networks:
    - prefix: 192.168.1.0/24

- hostname: R2
  dhcp_server:
    subnet: 192.168.2.0/24
    range: 192.168.2.2 192.168.2.254
  interfaces:
    - name: Ethernet1
      switch: 0 # no switchport
      ipv4_address: 172.20.10.1
      ipv4_mask: /31
    - name: Ethernet2
      switch: 0 # no switchport
      ipv4_address: 172.20.50.0
      ipv4_mask: /31
    - name: Ethernet3
      switch: 0 # no switchport
      ipv4_address: 192.168.2.1
      ipv4_mask: /24
      dhcp: true
    - name: Loopback0
      ipv4_address: 10.0.0.2
      ipv4_mask: /32
  ip_routing: true
  bgp:
    as_number: 55002
    router_id: 10.0.0.2
    neighbors:
      - ip: 172.20.10.0    # R1 - Ethernet1
        remote_as: 55001
      - ip: 172.20.50.1    # R3 - Ethernet1
        remote_as: 55010
    networks:
      - prefix: 192.168.2.0/24

```

```

- hostname: R3
  interfaces:
    - name: Ethernet1
      switch: 0 # no switchport
      ipv4_address: 172.20.50.1
      ipv4_mask: /31
    - name: Ethernet2
      switch: 0 # no switchport
      ipv4_address: 172.20.30.0
      ipv4_mask: /31
    - name: Loopback0
      ipv4_address: 10.0.0.3
      ipv4_mask: /32
  ip_routing: true
  bgp:
    as_number: 55010
    router_id: 10.0.0.3
    neighbors:
      - ip: 172.20.50.0 # R2 - Ethernet2
        remote_as: 55002

- hostname: R4
  interfaces:
    - name: Ethernet1
      switch: 0 # no switchport
      ipv4_address: 172.20.30.1
      ipv4_mask: /31
    - name: Ethernet2
      switch: 0 # no switchport
      ipv4_address: 172.20.40.0
      ipv4_mask: /31
    - name: Ethernet3
      switch: 0 # no switchport
      ipv4_address: 172.20.100.1
      ipv4_mask: /31
    - name: Loopback0
      ipv4_address: 10.0.0.4
      ipv4_mask: /32
  ip_routing: true
  bgp:
    as_number: 55020
    router_id: 10.0.0.4
    neighbors:
      - ip: 172.20.40.1 # R1 - Ethernet2
        remote_as: 55001
      - ip: 172.20.100.0 # internet_router
        remote_as: 54000
  networks:
    - prefix: 0.0.0.0/0

```

```

- hostname: internet_router
  interfaces:
    - name: Ethernet1
      switch: 0 # no switchport
      ipv4_address: 172.20.100.0
      ipv4_mask: /31 # Connection to R4
    - name: Ethernet2
      switch: 0 # no switchport
      ipv4_address: 128.128.128.1
      ipv4_mask: /24 # Connection to internet_host
    - name: Loopback0
      ipv4_address: 1.1.1.1
      ipv4_mask: /32 # Loopback to identify router
  ip_routing: true
  bgp:
    as_number: 54000
    router_id: 1.1.1.1
    neighbors:
      - ip: 172.20.100.1 # R4
        remote_as: 55020
    networks:
      - prefix: 128.128.128.0/24

- hostname: SW1
  interfaces:
    - name: Ethernet1
      switch: 1 # switchport
    - name: Ethernet2
      switch: 1 # switchport
    - name: Ethernet3
      switch: 1 # switchport
    - name: Loopback0
      ipv4_address: 10.0.1.1
      ipv4_mask: /32
  ip_routing: false

- hostname: SW2
  interfaces:
    - name: Ethernet1
      switch: 1 # switchport
    - name: Ethernet2
      switch: 1 # switchport
    - name: Ethernet3
      switch: 1 # switchport
    - name: Loopback0
      ipv4_address: 10.0.1.2
      ipv4_mask: /32
  ip_routing: false

```

## 5. Network Deployment Using Containerlab

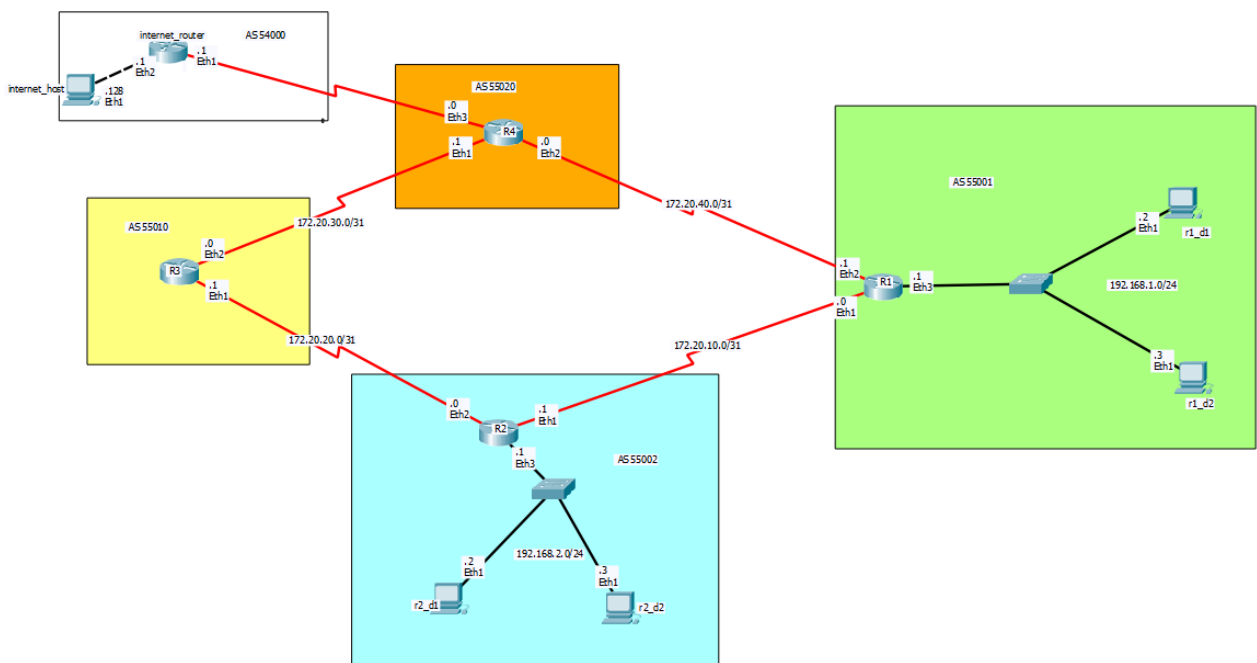
### 5.1 Network Topology Overview

The deployment process relies on a predefined network topology described previously in YAML format. Containerlab is used to instantiate virtual routers, switches, and hosts based on the defined topology.

Key components of the topology include:

- **Core Routers:** Establish backbone connectivity.
- **Access Switches:** Connect end devices to the network.
- **Hosts:** Simulate end-users or servers.

To carry out Task 2, the following topology has been created:



*This image has been generated by means of Cisco Packet Tracer.*

### 5.2 Deployment Process

#### 5.2.1 Main steps

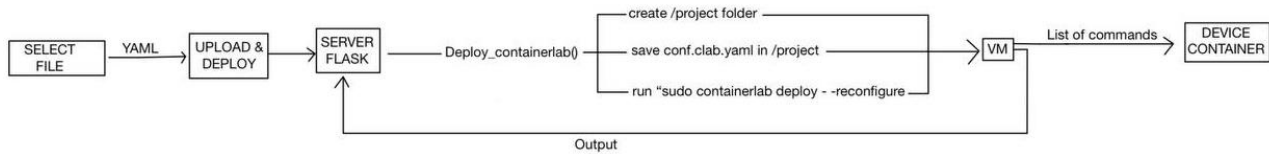
The deployment process using Containerlab involves the following steps:

1. **Node Creation:** After sending yaml file and the deploy command to the VM, Containerlab instantiates Arista containers for each device in the topology.
2. **Interconnection:** Virtual links are established based on the specified topology.

### 5.2.2 Detailed description

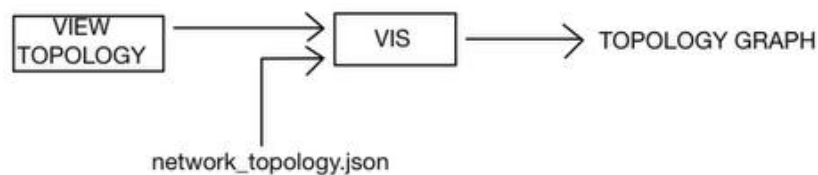
The core steps of the deployment process have been outlined. The following section provides a more detailed explanation of each phase, offering a comprehensive overview of the workflow.

After the yaml is uploaded and the deployment has been started, *flask server* creates a */project* folder in the VM and the yaml is saved there as *conf.clab.yaml*. Then, in the VM *sudo containerlab deploy - -reconfigure* is launched to start the network deployment. The *- -reconfigure* option is used to replace the network if it has been already created or if some containers haven't been destroyed. After the end of deployment, the operation status is shown on the page and the yaml is converted to json, saved as *network\_topology.json*. The following flow diagram shows these operations:



In addition, after the deployment of the network, the four configuration scripts are sent to the VM and saved in the *project* folder, in order to be executed in the next steps.

The just created topology can be seen in the GUI thanks to the proper button. When clicked, Vis library takes the json topology file as input and renders the network graph in a new web page. This is shown by this flow diagram:



### 5.3 Destroy Process

The created network can be destroyed by using the proper button in the GUI.

The following command is sent to the VM and executed:

```
sudo containerlab destroy && delete project
```

As can be seen, also the *project* folder is removed, to avoid errors during a new creation. The result of these operations is returned to the interface.

### 5.4 Internet Connectivity and Default Route Distribution

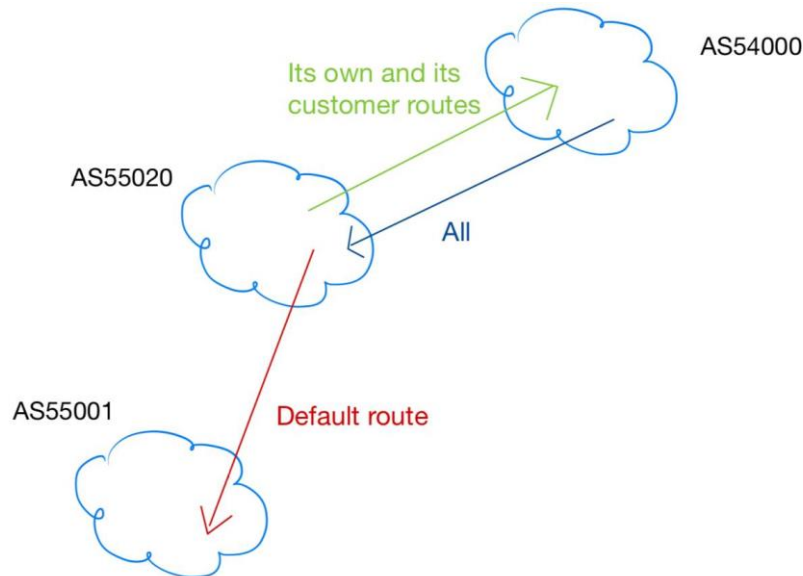
To expand the network's capabilities and simulate an internet connection, an extra router and host were added to the topology. This router, called "Internet Router," is connected to R4 (as shown in the topology image from the specs) and establishes a peer-to-peer relationship with R4. R4 shares its routes (none in this case) and its customers' routes with the Internet Router, while the latter advertises all the destinations it knows.

The Internet Router is set up to simulate a Tier 1 IXP, acting as an AS that peers with everyone and serves as a transit. The *internet\_host* connected to the Internet Router was added mainly for testing

purposes, letting us check simulated communication from hosts in the two LANs of the network to the internet.

R4 learns all the destinations from the Internet Router, but it only advertises a default route to its customers (like R1). So, in our network, the only AS with a full internet routing table (FIRT) is 55020, where R4 is located. This means that if any AS under R4 wants to access the internet, the packets are sent to R4 via the default route. Then, since R4 has the FIRT, it can forward those packets to any destination.

Below is an image summarizing the BGP relationships between the Internet Router and R4:



## 6. Automated Network Testing

### 6.1 Testing Tools: iperf and tcpdump

Testing the deployed network involves the use of the following tools:

- **iperf**: Measures network bandwidth between hosts.
- **tcpdump**: Captures and analyzes network traffic for troubleshooting and validation.

Command format to run an **iperf** test:

- *iperf -c <destination address> -i 2 -u* (on client)
- *iperf -u -s -p <port> -t <timeout>*

Options:

- *-c* → specifies that this host is the client.
- *-s* → specifies that this host is the server
- *-i 2* → defines the interval in seconds between periodic reports.
- *-u* → specifies udp/tcp protocol (is default).
- *destination address* → defines the receiver ip address.
- *-t* → server timeout, after that server terminates
- *-p* → indicates the receiving port.

Example of using **tcpdump** to capture packets incoming in a specific interface:

- *tcpdump -i eth0*

### 6.2 Testing Scenarios and Results

Two scenarios were tested to validate network performance:

#### Scenario 1: End-to-End Connectivity

- Test: Verify that all devices in the topology can communicate.
- Result: Successful pings between all devices.

#### Scenario 2: Packet Analysis

- Test: Use *tcpdump* to inspect packets for routing accuracy.
- Result: All packets were routed correctly without drops.

Two hosts can be selected between all hosts in the virtual network, together with a device and one of its own interfaces to carry out these scenarios. This information is gathered creating a configuration that will be sent to a remote script on the VM. This script is triggered by the GUI and will perform *iperf* and *tcpdump* commands on devices, returning their outputs to the interface.

This is the command run on the VM to start client iperf in a host container:



***docker exec client\_hostname iperf -c server\_ip -i tentatives transport\_protocol\_option -p port -w 256k***

Parameters:

- *client\_hostname* → name of the host device
- *server\_ip* → name of the host device where the server is executed
- *tentatives* → the interval in seconds between periodic reports
- *transport\_protocol\_option* → chosen transport protocol
- *port* → receiving port (fixed to 62000)
- *-w 256k* → dimension of the TCP window

This is the command run on the VM to start iperf server in a host container:

***docker exec server\_hostname iperf transport\_protocol\_option -s -p port -t timeout***

Parameters:

- *server\_hostname* → name of the host device
- *transport\_protocol\_option* → chosen transport protocol
- *port* → receiving port (fixed to 62000)
- *timeout* → time server will wait before terminating (fixed to 12s)

This is the command run on the VM to start tcpdump in a device container:

***docker exec tcpdump\_device\_name timeout timeouts tcpdump -i tcpdump\_device\_interface tcpdump\_filter port port -w /dev/null -c tcpdump\_maximum\_number\_of\_packets***

Parameters:

- *tcpdump\_device\_name* → name of the chosen device
- *timeout* → time tcpdump is executed (fixed to 12s)
- *tcpdump\_filter* → transport protocol filter
- *tcpdump\_device\_interface* → chosen device interface
- *port* → receiving port (fixed to 62000)
- *-w /dev/null* → data are deleted after the capture
- *tcpdump\_maximum\_number\_of\_packets* → set a max number of packets captured (fixed to 20)

Example of an iperf client output:

--- CLIENT RESULT ---

Host clab-network\_topology-rl\_d1 (192.168.1.2): Success

Client connecting to 192.168.1.3, TCP port 62000

TCP window size: 416 KByte (WARNING: requested 250 KByte)

```
[ 1] local 192.168.1.2 port 59402 connected with 192.168.1.3 port 62000
[ ID] Interval          Transfer      Bandwidth
[ 1] 0.00-2.00 sec      415 KBytes   1.70 Mbits/sec
```

```
[ 1] 2.00-4.00 sec  0.000 Bytes  0.000 bits/sec
[ 1] 4.00-6.00 sec  0.000 Bytes  0.000 bits/sec
[ 1] tcp write (1736777795.950392) failed
[ 1] shutdown failed: Socket not connected
[ 1] 0.00-7.91 sec   415 KBytes   430 Kbits/sec
```

Example of an iperf server output:

```
--- SERVER RESULT ---
```

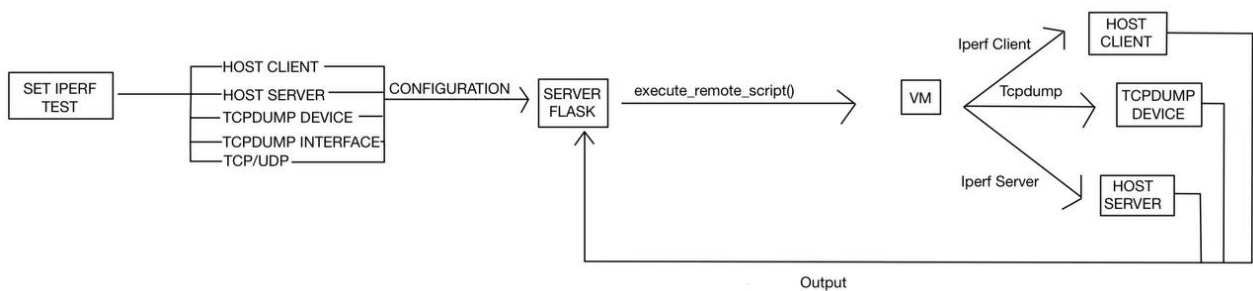
```
Host clab-network_topology-r1_d2 (192.168.1.3): Server finished after 12
seconds
```

Example of an tcpdump output:

```
--- TCPDUMP RESULT ---
```

```
Host clab-network_topology-r2 (eth3): Success
tcpdump: listening on eth3, link-type EN10MB (Ethernet), snapshot length
262144 bytes
20 packets captured
93 packets received by filter
0 packets dropped by kernel
```

The following flow diagram illustrates the explained operations:



## 7. Graphical User Interface (GUI)

### 7.1 Features and Functionalities

The Graphical User Interface (GUI) provides an intuitive and user-friendly way to manage the network automation and deployment process.

Key features include:

- **Deployment Automation:** Initiate deployment loading topology yaml file directly from the web-interface.
- **Topology Visualization:** Display the deployed network topology in a graphical format to check if the yaml topology is correct (this is done by converting yaml to json that will be used as input in a specific function).
- **Configuration Management:** Load and send network configurations.
- **Real-Time Monitoring:** Test traffic and communication between network devices.

### 7.2 Workflow Explanation

The GUI follows a streamlined workflow to simplify network management:

1. **Deployment**
  - The user clicks "Upload and Deploy" button to initiate network deployment using Containerlab after having uploaded the yaml topology file.
  - When the deployment is finished, a message appears on the screen.
2. **Load Configuration**
  - The user uploads a YAML file for device configuration.
  - The GUI sends it to virtual network devices.
3. **Topology Visualization**
  - The uploaded topology is converted into a graphical representation of the network.
  - Devices and links are displayed.
4. **Monitoring and Testing**
  - The GUI provides an interface for running **iperf** between two devices and visualizing traffic with **tcpdump** in a specific interface of a third device.
  - Command outputs are shown in the interface after completion, in order to visualize the traffic.

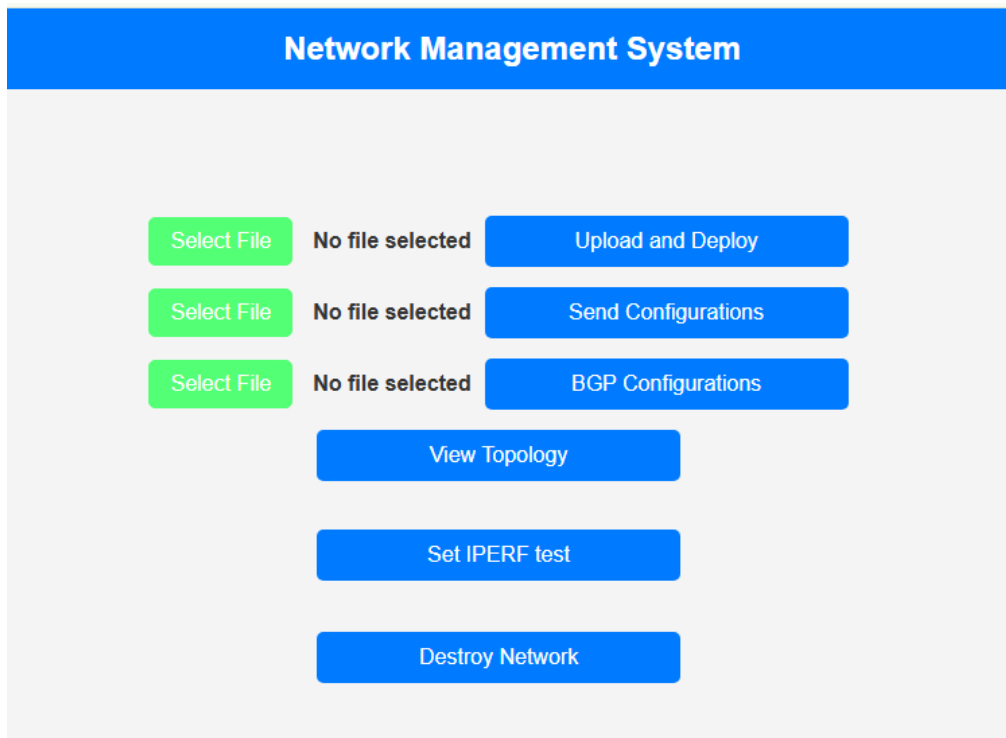
## 7.3 User Guide

This section explains how to use the GUI effectively:

### 7.3.1 Accessing the GUI

- Open the GUI in a web browser running *index.html*  
(e.g. [http://localhost:63342/ANAWS\\_Project/web\\_interface/index.html](http://localhost:63342/ANAWS_Project/web_interface/index.html)).

This is the main page:

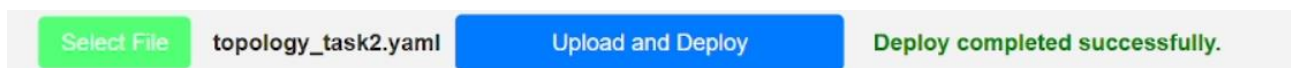


### 7.3.2 Performing Deployment

1. Click on the first "Select File" from top.
2. Select a topology yaml file.

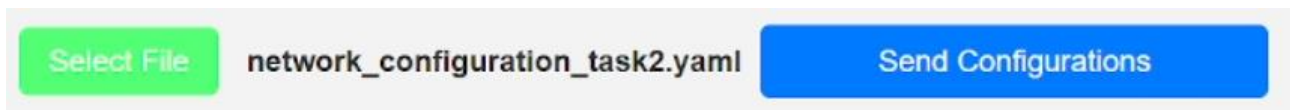


3. When finished, a green message will appear on the screen:

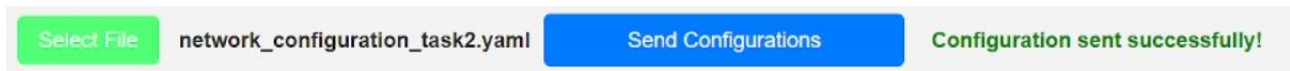


### 7.3.3 Performing Routers and Switches configuration

1. Click on the second “Select File” from top.
2. Select a configuration yaml file.

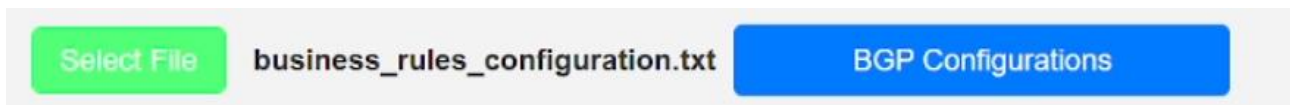


3. When finished, a green message will appear on the screen:

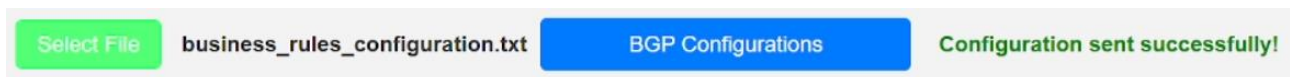


### 7.3.4 Performing BGP configuration

1. Click on the third “Select File” from top.
2. Select a BGP configuration txt file.

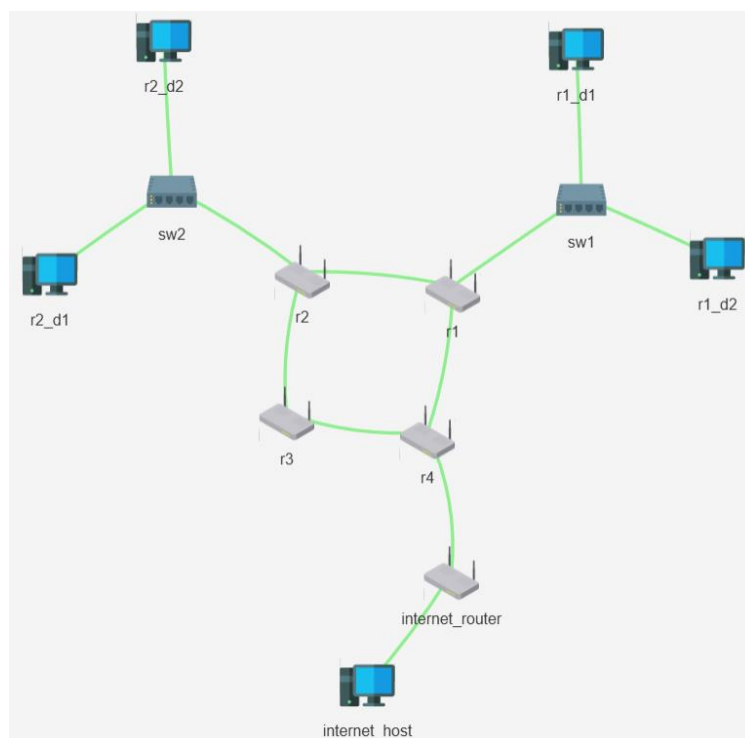


3. When finished, a green message will appear on the screen:



### 7.3.5 View created topology

1. Click on “View Topology”
2. The topology will be shown, like in the following case:



### 7.3.4 Performing Iperf Test

1. Click on “Set IPERF Test”
2. Information on routers and hosts is taken, during the operation the following message is shown:

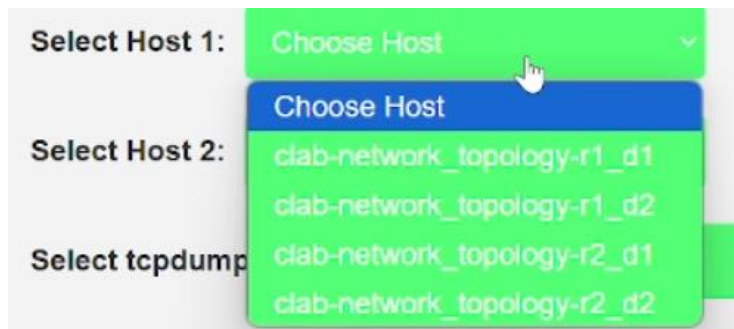
Waiting for the fetching operation...

3. After fetching, the following input mask appears:



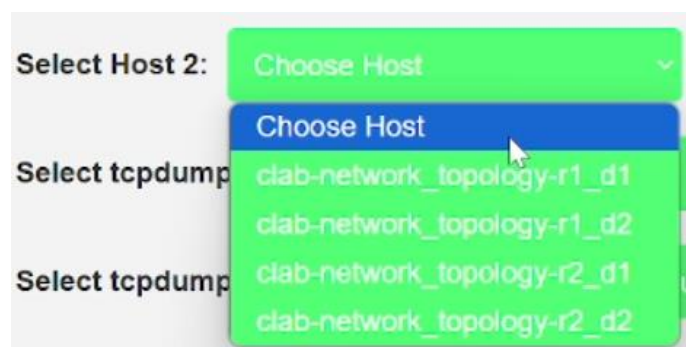
The screenshot shows a configuration form for the Iperf Test. It contains five dropdown menus and a submit button. The dropdowns are labeled: 'Select Host 1:', 'Select Host 2:', 'Select tcpdump device:', 'Select tcpdump device interface:', and 'Select traffic type:'. The 'Select Host 1:' and 'Select Host 2:' dropdowns show 'Choose Host'. The 'Select tcpdump device:' dropdown shows 'Choose Router'. The 'Select tcpdump device interface:' dropdown shows 'Choose tcpdump device inter'. The 'Select traffic type:' dropdown shows 'TCP'. A blue 'Submit' button is at the bottom.

4. Choose Host 1 between those available:



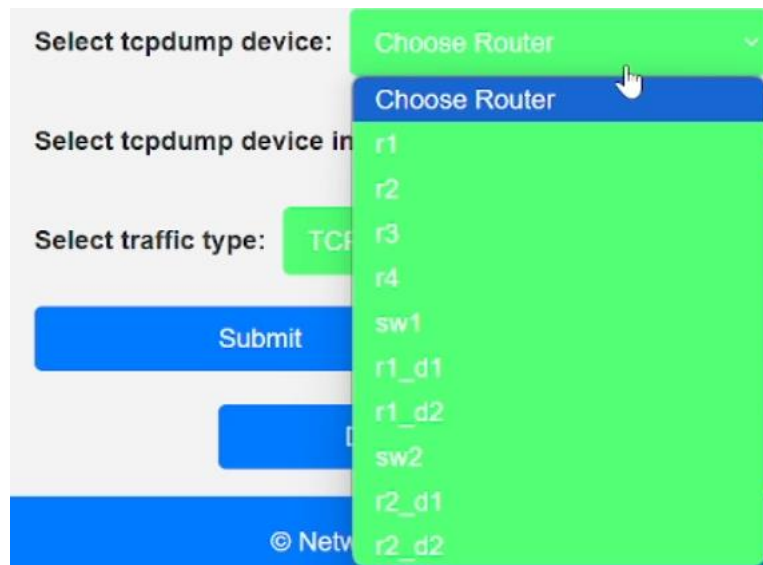
The screenshot shows the 'Select Host 1:' dropdown menu open. The dropdown list contains the following options: 'Choose Host', 'clab-network\_topology-r1\_d1', 'clab-network\_topology-r1\_d2', 'clab-network\_topology-r2\_d1', and 'clab-network\_topology-r2\_d2'. A mouse cursor is pointing at the 'Choose Host' option.

5. Choose Host 2 between those available:



The screenshot shows the 'Select Host 2:' dropdown menu open. The dropdown list contains the following options: 'Choose Host', 'clab-network\_topology-r1\_d1', 'clab-network\_topology-r1\_d2', 'clab-network\_topology-r2\_d1', and 'clab-network\_topology-r2\_d2'. A mouse cursor is pointing at the 'Choose Host' option.

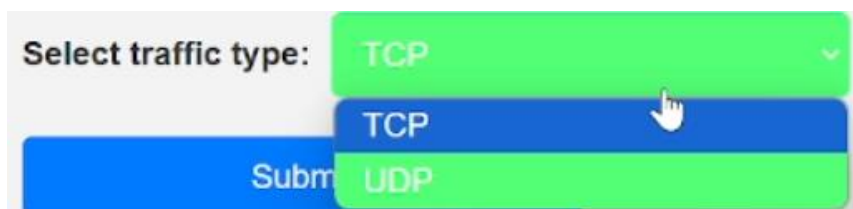
6. Choose tcpdump device between those available:



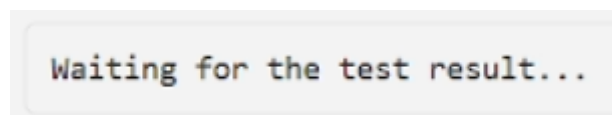
7. Choose an interface of that device to monitor:



8. Choose traffic type:



9. After clicking on “Submit” button, a message will appear:



10. After all operations have been completed, the results are shown:

```

--- CLIENT RESULT ---
Host clab-network_topology-r1_d1 (192.168.1.2): Success
-----
Client connecting to 192.168.2.2, UDP port 62000
Sending 1470 byte datagrams, IPG target: 0.00 us (kalman adjust)
UDP buffer size: 416 KByte (WARNING: requested 250 KByte)
-----
[ 1] local 192.168.1.2 port 59823 connected with 192.168.2.2 port 62000
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.00-2.00 sec   258 KBytes    1.06 Mbits/sec
[ 1] 2.00-4.00 sec   256 KBytes    1.05 Mbits/sec
[ 1] 4.00-6.00 sec   256 KBytes    1.05 Mbits/sec
[ 1] 6.00-8.00 sec   257 KBytes    1.05 Mbits/sec
[ 1] 8.00-10.00 sec  256 KBytes    1.05 Mbits/sec
[ 1] 0.00-10.02 sec  1.25 MBytes   1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.00-9.98 sec   1.25 MBytes   1.05 Mbits/sec  0.000 ms  1/895 (0%)

```

```

--- SERVER RESULT ---
Host clab-network_topology-r2_d1 (192.168.2.2): Server finished after 12 seconds

```

```

--- TCPDUMP RESULT ---
Host clab-network_topology-r1 (eth3): Success
tcpdump: listening on eth3, link-type EN10MB (Ethernet), snapshot length 262144 bytes
20 packets captured
91 packets received by filter
0 packets dropped by kernel

```

### 7.3.4 Performing Network Destruction

Click on the following button:

Destroy Network

A message of completion will appear.



## 8. Conclusion

This project successfully achieved its primary objectives, demonstrating the ability to:

- Automate network configuration using YAML and txt based templates for DHCP and BGP.
- Deploy complex network topologies using Containerlab starting from a YAML topology file.
- Provide a user-friendly GUI for managing, monitoring, and testing the network.
- Validate network performance through automated testing tools like **iperf** and **tcpdump**.

These advancements showcase the power of automation in network management and its potential for scalability and efficiency in real-world applications.